厳選 0b1010 個!
C99/C11/C++11/C++14/C++17 の
今すぐ使えるコーディング Tips と応用事例

2018-12-01

株式会社ポリフォニー・デジタル 福岡アトリエ 山之内 毅

takeshi_yamanouchi@polyphony.co.jp

はじめに

- 本セッションでは、C と C++ を楽しく便利に書ける Tips を紹介します。

- ゲーム製作に有用と考えられるものを選びました。

- コードのテストは gcc 7.3.0 で行なっています。

- 10個のTipsのうち、実タイトルへの応用事例は 4番目でご紹介します。

プログラマでない方、もしいらっしゃいましたら、少々お付き合いください。

- それでは早速、セッションタイトルの Tips からはじめましょう!

1. 数値定数(C++14, C99/C++17)

- 2進数定数(C++14)
  - 16進数の0xのように0bで始めると2進数表記。
  - シングルクォートが区切り記号(セパレーター)。
  - 2進数以外や float, double でも区切り記号は使えます。

※UTF-8文字列定数(C++11)

u8を前につけると UTF-8 エンコードになります。ない場合は処理系依存。

→ ここでは以降、特殊な記号以外の u8 を省略します。

```
printf(u8"厳選 %d 個!\n", 0b1010); // 10
       // 文字列定数前にu8をつけるとUTF-8
int a = 0b1001'1100'0100;          // 0x9c4
printf("%c%c%d\n", 0b01'101'101,  // 0155
                   0b01'111'010,  // 0172
                   a);

// セパレータは何箇所に挟んでも構わない
int    b = 1'23'4567'89;
double c = 1'2'3'4'5'6.7'8'9;
printf("%d %f\n", b, c);

[実行結果]
厳選 10 個!
mz2500
123456789 123456.789000
```

1. 数値定数(C++14, C99/C++17)

- 16進数浮動小数点定数(C99/C++17)
  - 「0.01 を 10000 回足しても 100 にならない」ことが最近話題に。
    → しばしば 2進数で割り切れる数を書きたい。1/8 とか 25/32 とか。
  - 0xで始め、仮数部を16進、pの後に、指数部を10進で書く。指数部の底は2。
  - pで始まる指数部の省略は不可。

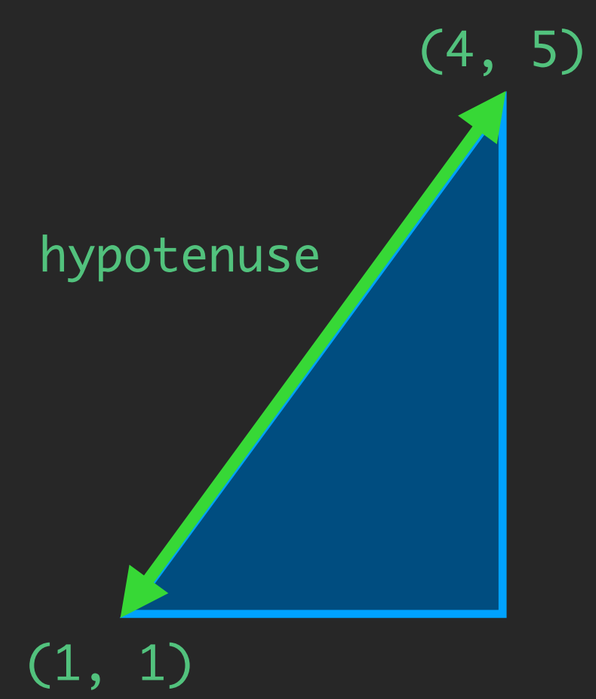
```
float  d = 0x1.0p-5f;  // 1/32
float  e = 0xA.aP+0f;  // 10+10/16  大文字小文字の混在可
double f = 0x1p+24;    // 16777216  小数部の省略可
printf("%f %f %f\n", d, e, f);

// 10進数の 0.01 は 2進数で循環小数 // 0.0000'0010'1000'1111'0101'1100...
printf("%.20f %a [%c]\n", 0.01, 0.01, (0.01 == 0x0.0'28f5c'28f5c'28f5cp+0) ? 'T' : 'F');

[実行結果]
0.031250 10.625000 16777216.000000
0.01000000000000000021 0x1.47ae147ae147bp-7 [T]
```

2. hypot(C99/C++17)

- 三角形の斜辺(hypotenuse) の長さを求める関数(C99)
  - 1フレームに一度の箇所やツールなど、速度をかつかつに求めない場合に有用。
  - 2次元のみ(cabsと同じ)。
  - hypotf (float)
  - hypotl (long double)




```
#include <math.h>

double x0 = 1;
double y0 = 1;
double x1 = 4;
double y1 = 5;

// 2乗で 2回書くので気が萎える
double r0 = sqrt((x1-x0)*(x1-x0) + (y1-y0)*(y1-y0));

// 気楽!
double r1 = hypot(x1-x0, y1-y0);
printf("hypot %f %f\n", r0, r1);

[実行結果]
hypot 5.000000 5.000000
```

2. hypot(C99/C++17)

- 3 次元版 hypot(C++17)
  - C++17に導入の際、3引数の 3次元版が追加。
  - float, long double 版も、引数違いの関数オーバーロードで定義されています。

```
#include <cmath>

y1 = 9;
double z0 = 1;
double z1 = 37;
double r4 = std::hypot(x1-x0, y1-y0, z1-z0);

printf("hypot3D %f\n", r4);
```

[実行結果]

```
hypot3D 37.000000
```

3. aligned_alloc(C11/C++17)

- アラインメント付き、動的メモリの確保

  ゲーム開発では、しばしば vec4 のような SIMD の型を用い、メモリ確保のアラインメントが問題になります。

- 従来の関数の例

  - posix_memalign

    UNIX 共通の規格。

    返値が確保したポインターではなく、成否なのが微妙に使い辛い。

  - _mm_malloc

    インテル CPU で SIMD(SSE, AVX) を使う場合の標準。

    free ではなく、_mm_free で解放。

- C11/C++17 で言語標準の関数ができました!

  → aligned_alloc

3. aligned_alloc(C11/C++17)

- aligned_alloc 例

```
#include <stdlib.h>
#include <cstdlib>

size_t sz = 1024;
size_t al = 256;  // sz は al の整数倍であること
void *r0 =      malloc(sz);
void *r1 = std::malloc(sz);
void *r2 =      aligned_alloc(al, sz);
void *r3 = std::aligned_alloc(al, sz);

printf("malloc         %p %p\n", r0, r1);
printf("aligned_alloc %p %p\n", r2, r3);
     free(r0);
std::free(r1);
     free(r2);
std::free(r3);
```

[実行結果]

```
malloc         0xd83c10 0xd84020
aligned_alloc 0xd84500 0xd84a00
                    ↑        ↑ 256 バイト単位で揃う
```

3. aligned_alloc(C11/C++17)

- ところで、静的変数にアラインメントを指定したい場合
  - 従来はコンパイラ固有の属性で指定。
  - C++11 で、alignas 指定子が追加。

    → 静的確保には良かったものの、動的確保には何の影響もありませんでした…。

```
const size_t al = 256;                          // 定数であること
int                                  r0, r1;    // align 指定なし
int __attribute__((aligned(al))) r2;            // gcc の属性で指定
alignas(al) int                      r3;        // C++11 指定子

printf("stack         %p %p\n", &r0, &r1);
printf("aligned stack %p %p\n", &r2, &r3);

[実行結果]
stack         0x7ffc6bc1a5b0 0x7ffc6bc1a5b4
aligned stack 0x7ffc6bc1a400 0x7ffc6bc1a500
                           ↑              ↑ 256 バイト単位で揃う
```

3. aligned_alloc(C11/C++17)

- アラインメント付き new, delete(C++17)
  - C++17では、new, delete にアラインメント(std::align_val_t) を引数にとるオーバーロードが追加。alignas の指定が(適切に)引き渡されます。

```
struct                 vec4          { float x, y, z, w; };
struct                 mat4          { vec4 column[4]; };
struct alignas(16)     vec4_aligned  { float x, y, z, w; };
struct alignas(16*4)   mat4_aligned  { vec4_aligned column[4]; };

auto *m0 = new mat4 [3];
auto *m1 = new mat4 [3];
auto *m2 = new mat4_aligned [3];
auto *m3 = new mat4_aligned [3];
printf("new         %p %p\n", m0, m1);
printf("aligned new %p %p\n", m2, m3);
delete [] m0;  // m1..m3 省略

std::vector<mat4_aligned> mat4_array(10);  // STL コンテナにも渡せる
printf("aligned container %p %p\n", &mat4_array[0], &mat4_array[1]);
```

3. aligned_alloc(C11/C++17)

- アラインメント付き new, delete 実行結果
  - alignas の指定が動的メモリ確保にも効くようになりました。

```
[実行結果]
new           0xbe3430 0xbe3eb0
aligned new 0xbe3f80 0xbe40c0
                    ↑        ↑ 64 バイト単位で揃う

aligned container 0xbe4200 0xbe4240
                             ↑        ↑ 64 バイト単位で揃う
```

4. 乱数ライブラリー(C++11)

従来 rand() 等で直接整数の乱数を得ていましたが、
C++11 で、乱数エンジンと乱数分布に分離しました。

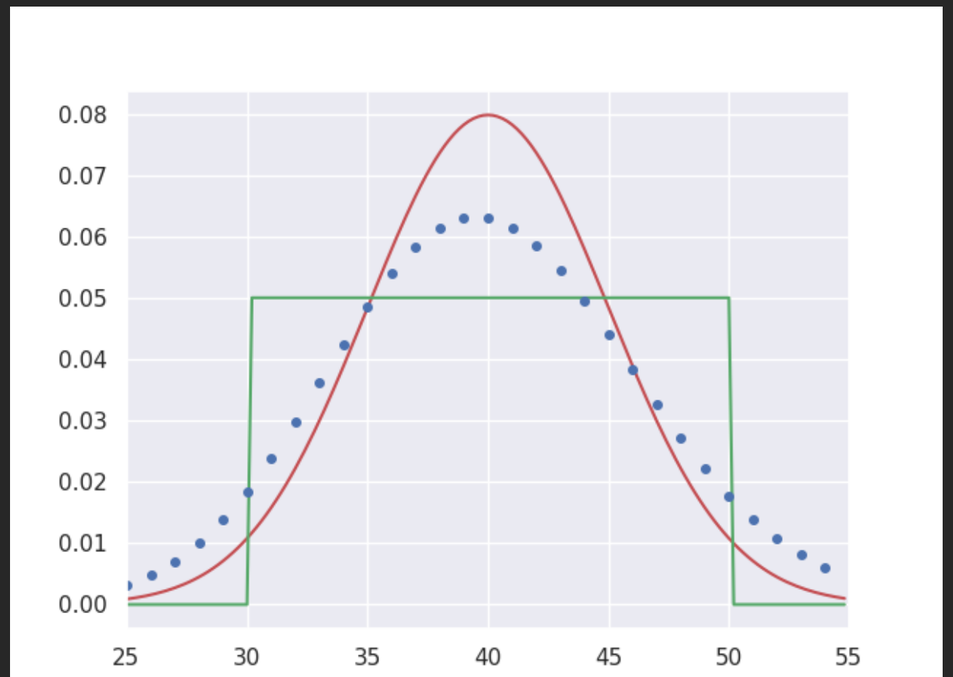
- 乱数エンジン
  - 符号無し整数値を返す、一様擬似乱数生成器(URBG: Uniform Random Bit Generator)。
  - メルセンヌツイスターは、およそ 5KB あります。スタックにとる場合は注意。

```
#include <random>
// 乱数デバイス(シード設定用)
std::random_device dev;

// 乱数生成器
std::mt19937 gen_mt();    //シード設定なし時はデフォルトのシードが使われる
std::default_random_engine gen_def(dev());
```

4. 乱数ライブラリー(C++11)

- 乱数分布
  - 乱数エンジンの出力を、望みの分布に従うよう加工。
  - 一様分布、正規分布、ポアソン分布など多数用意されています。



乱数分布の例

```
// 一様分布(指定範囲内に均等に出現する分布)に従う乱数
std::uniform_real_distribution<double> distr_double(0.0, 1.0);
std::uniform_int_distribution<>        distr_int(-10, 10);
printf("random %f %d\n", distr_double(gen_mt), distr_int(gen_def));

[実行結果]
random 0.397529 -4
```

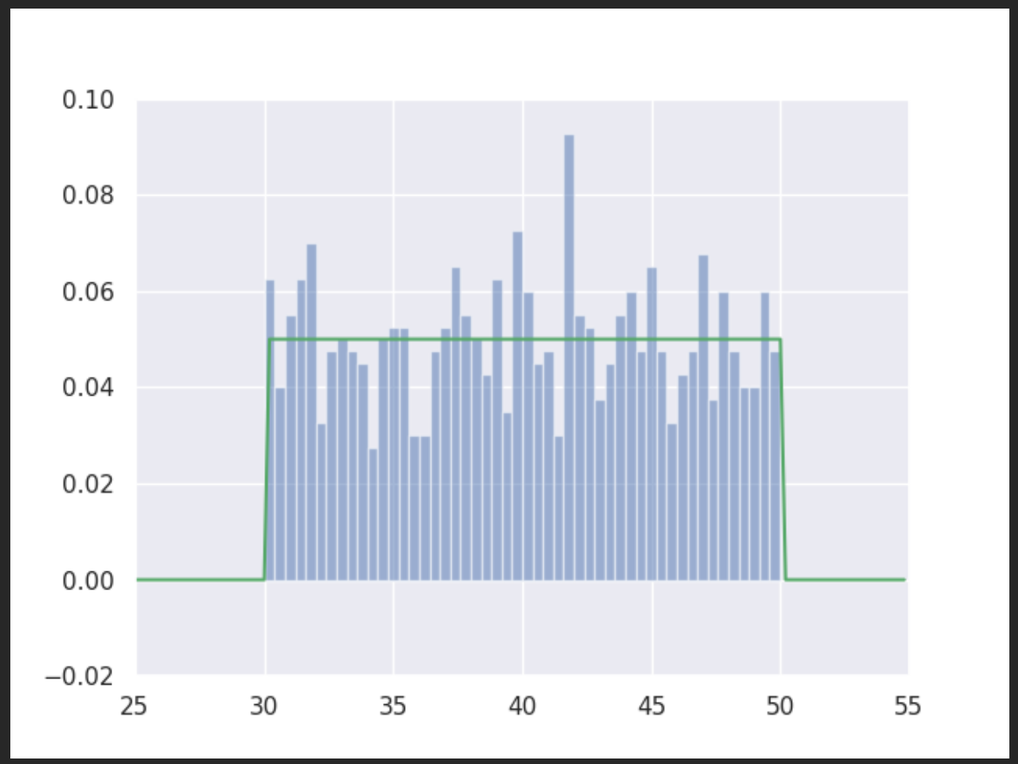
4. 乱数ライブラリー(C++11)

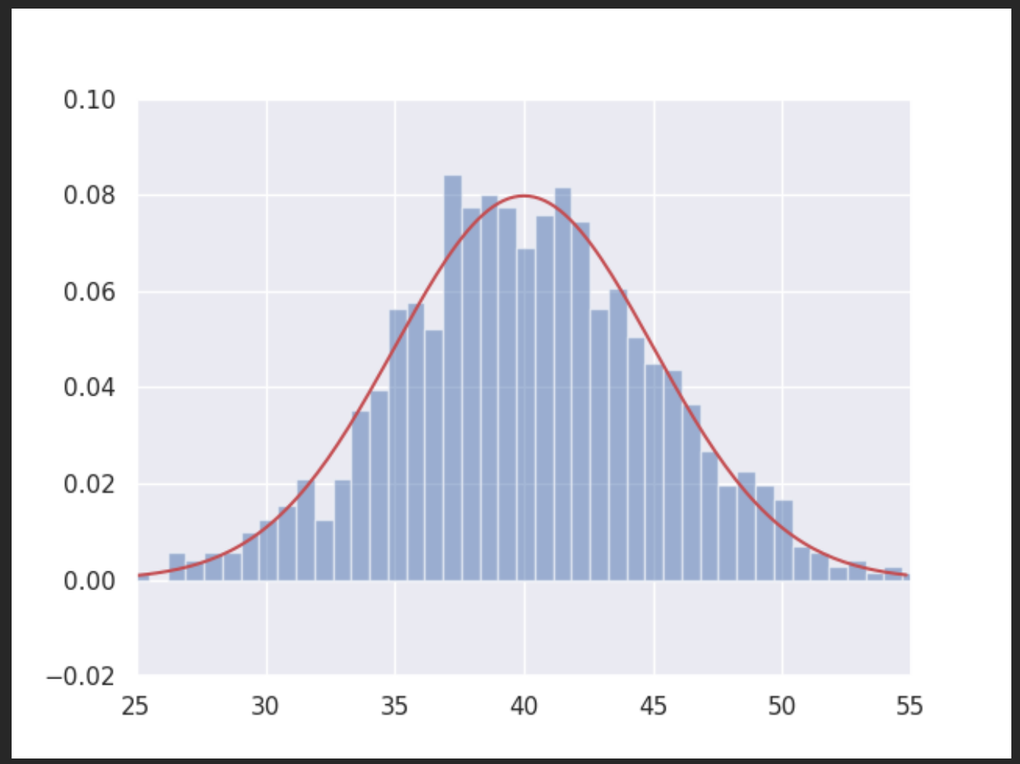
- 乱数分布の例。一様分布と正規分布の比較

  生物の体長などが、正規分布に従うことが知られています。

  標準偏差σ、±σ内に約68%、±2σ内に約95%、±3σ内に約99.7%

```
std::normal_distribution<double> distr_normal (40, 5);  // 平均 40, 標準偏差 5 の分布
```





一様分布と正規分布に従って発生させた乱数のヒストグラム

4. 乱数ライブラリー(C++11)

```
std::default_random_engine gen;
// 乱数テスト
// distr の型が異なるので、auto(多相ラムダ)で受け取る
auto test_random = [&] (const char *mes, int n, auto distr) {
  puts(mes);
  double mean = 0;              // ここで発生させる乱数の平均と
  double vari = 0;              // 分散を集計
  for (int i=0; i<n; i++) {
    double r = distr(gen);      // 乱数生成
    mean += r;
    vari += r * r;
    puts(std::string((r+0.5), '#').c_str());
  }
  mean = (mean / n);
  vari = (vari / n) - (mean * mean);
  printf("mean %f stddev %f\n", mean, sqrt(vari));
};
// 一様分布と正規分布
std::uniform_real_distribution<double> distr_uniform(30, 50);   // 最小、最大
std::normal_distribution<double>       distr_normal (40, 5.77); // 平均、標準偏差
int n = 50;    // 発生回数
test_random("uniform distribution ---", n, distr_uniform);
test_random("normal distribution ---" , n, distr_normal);
```

- 左: 一様分布
- 右: 正規分布

  右の方が、平均に近い値が多く生成されているように見えます。

```
uniform distribution ---
##############################
####################################
################################
########################################
##############################################
####################################
############################
##################################
############################
############################################
##############################################
####################################
########################################
############################
##################################
############################################
####################################
##############################################
########################################
############################
##############################################
####################################
########################################
##################################
########################################
##############################################
##################################
########################################
####################################
########################################
mean 39.756577 stddev 5.942478
```

```
normal distribution ---
##############################
####################################
################################
########################################
##############################################
####################################
############################
##################################
############################
############################################
##############################################
############################
##################################
############################################
####################################
##############################################
############################
##########################################
########################################
############################
##########################################
####################################
########################################
##################################
########################################
##############################################
##################################
##########################################
############################
##########################################
####################################
########################################
mean 39.806702 stddev 5.200046
```

4. 乱数ライブラリー(C++11)

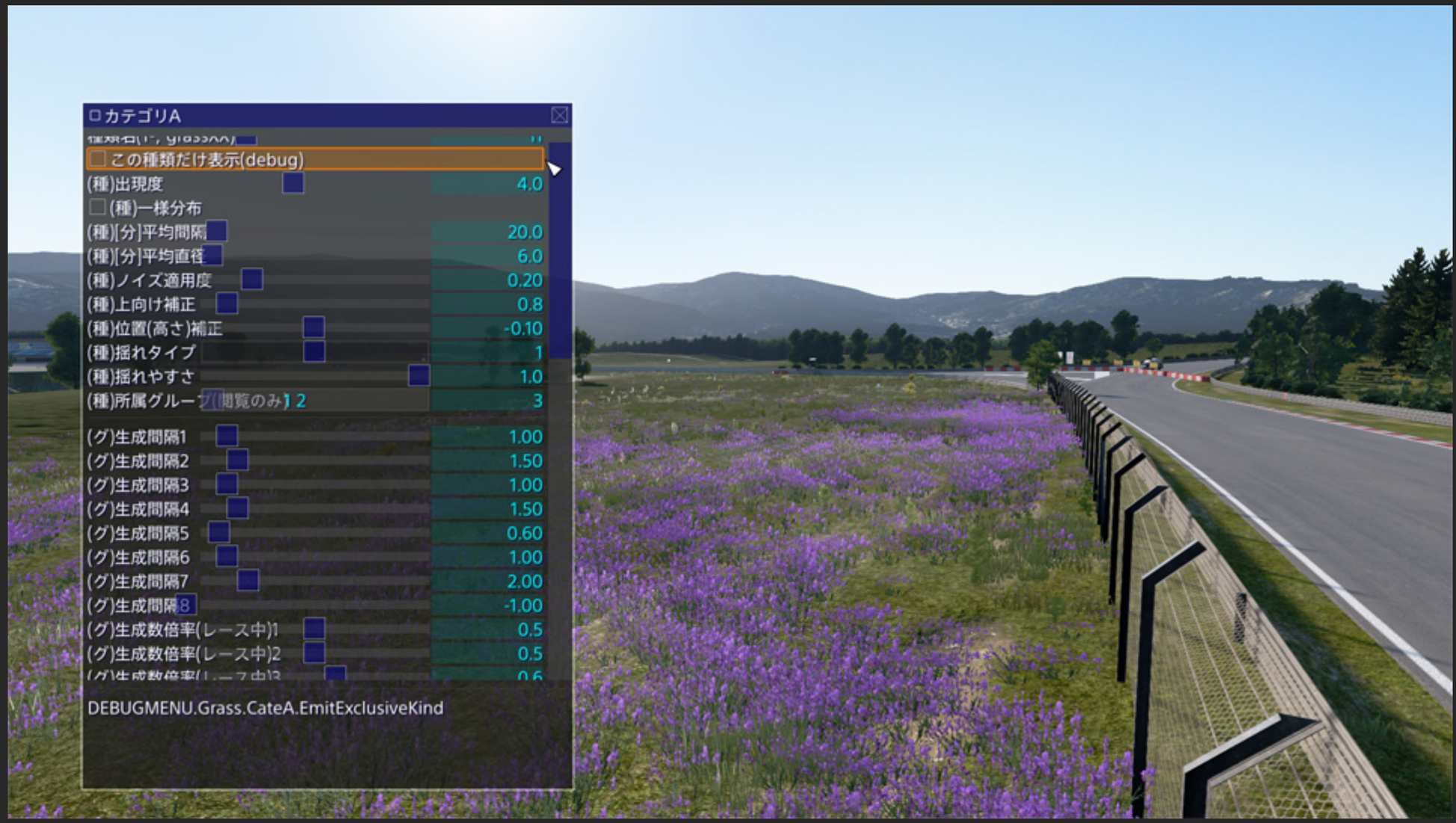
- 生成させた乱数の平均と標準偏差
  - 多数生成させると、分布に従っているのがわかります。

```
std::uniform_real_distribution<double> distr_uniform(30, 50);   // 最小、最大
std::normal_distribution<double>       distr_normal (40, 5.77); // 平均、標準偏差
30..50 の 一様分布の平均と標準偏差は次の通り
平均     (50-30)/2           = 40
標準偏差 sqrt((50-30)^2/12) =  5.7735

[実行結果、n = 50 の時の平均と標準偏差]
uniform distribution ---
mean 39.756577 stddev 5.942478
normal distribution ---
mean 39.806702 stddev 5.200046
[実行結果、n = 10000 の時の平均と標準偏差]
uniform distribution ---
mean 40.001530 stddev 5.763824
normal distribution ---
mean 39.971534 stddev 5.746089
```

4. 乱数ライブラリー(C++11) (応用例)

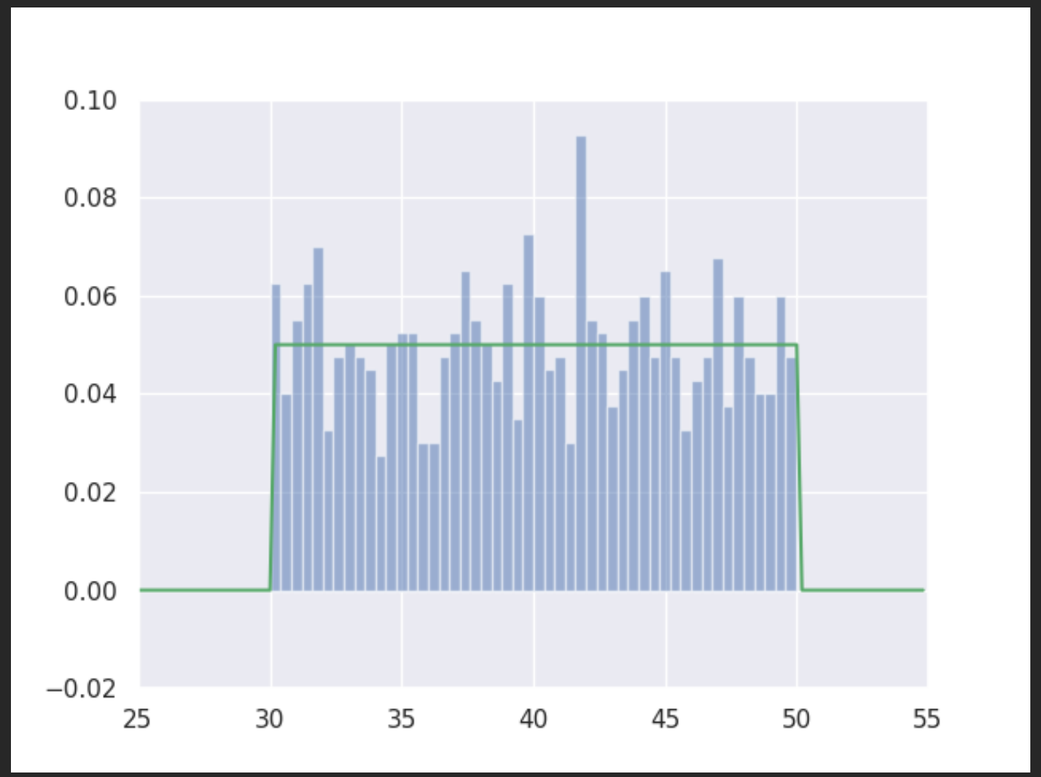
- 手続き的に生成するオブジェクトへの応用例

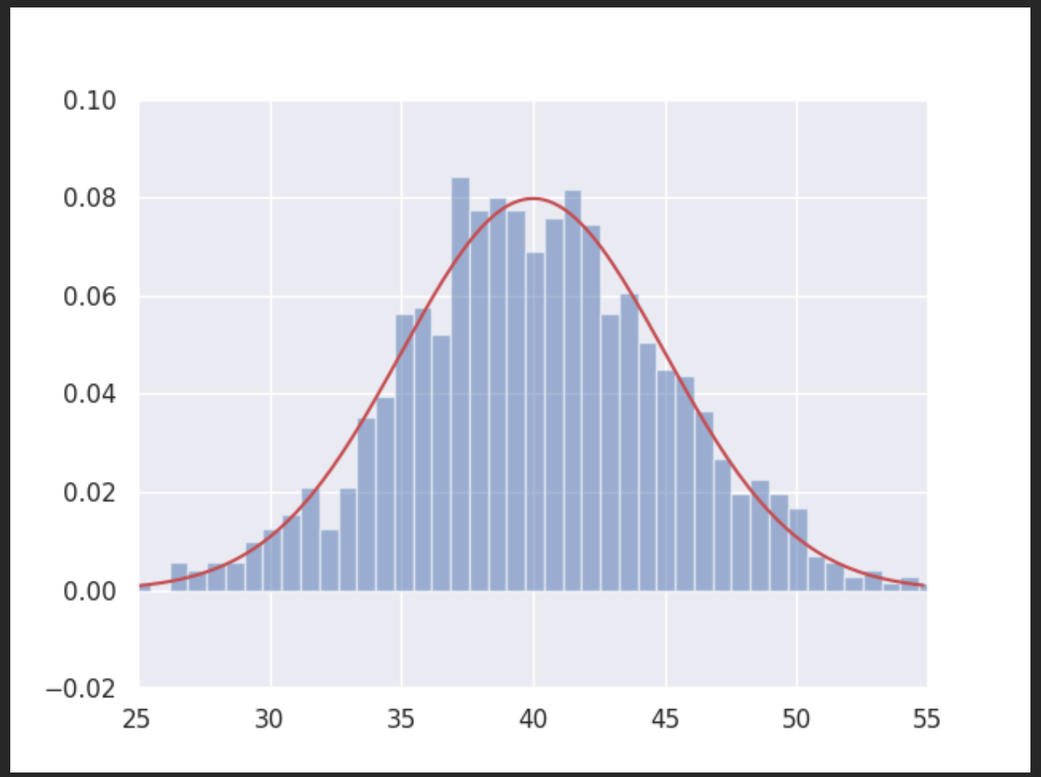


パラメータに従って、多数の草や花を自動生成

4. 乱数ライブラリー(C++11) (応用例)

- 生物の体長は、正規分布に従う典型的な例
  - ビルボードの草花のサイズをばらつかせるのに好適では?
  - 一様分布と正規分布で、サイズを変化させてみましょう。





乱数のヒストグラム再掲。これらの分布でサイズを変えてみます。

4. 乱数ライブラリー(C++11) (応用例)

- 一様分布でサイズを変化

凸凹している印象

4. 乱数ライブラリー(C++11) (応用例)

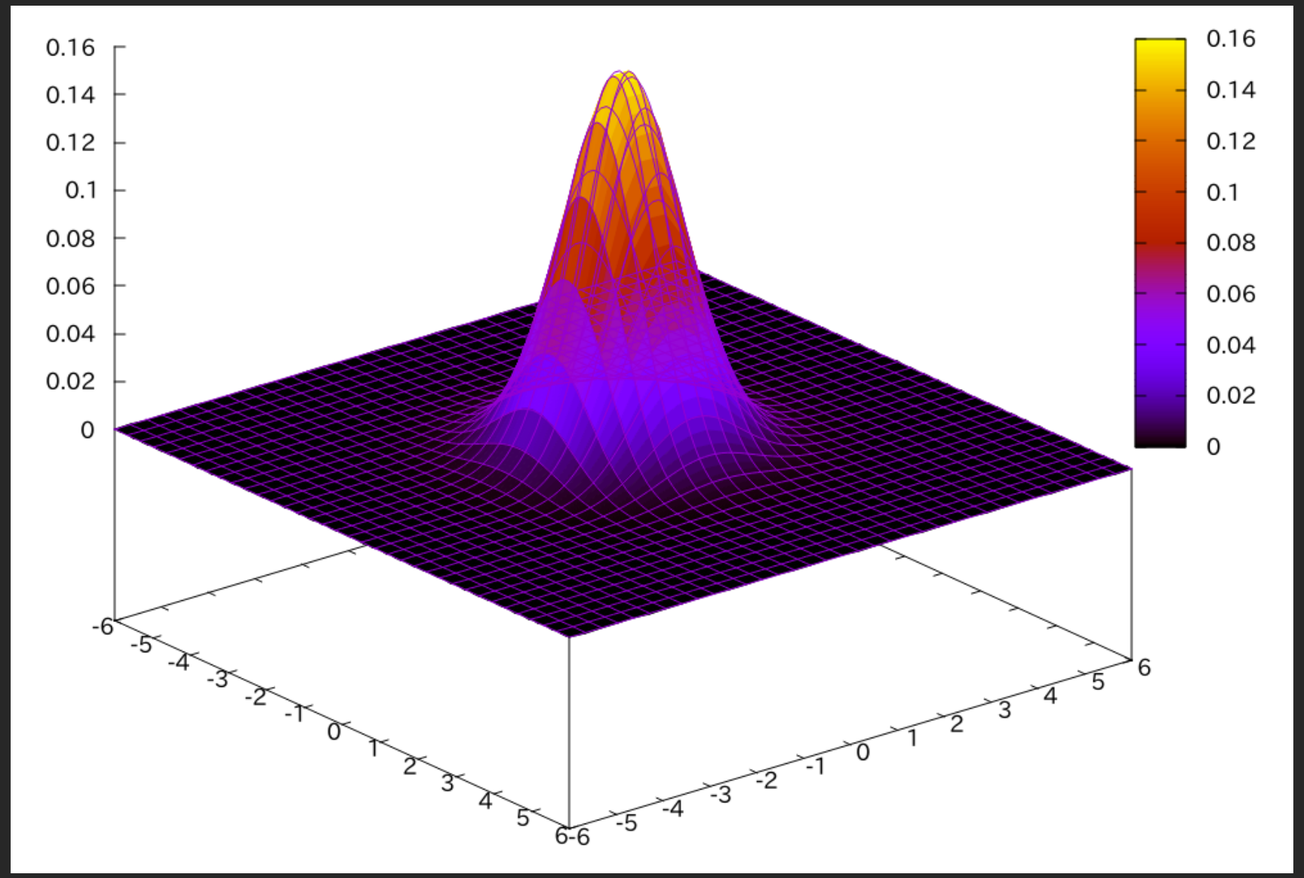
- 正規分布でサイズを変化

平均に近いサイズの花が多くなり自然な印象

4. 乱数ライブラリー(C++11) (応用例)

- 多変量正規分布
  - 2次元正規分布が、草花の配置座標生成に使えるのでは?

群生で、中心に多数、周辺に行くに従って減少してほしい。(注意：統計的な根拠はなし)



2次元正規分布のグラフ

4. 乱数ライブラリー(C++11) (応用例)

- 2次元正規分布で配置

ちょっと思ってたのと違う…シニヨンとかマカロンぽい。

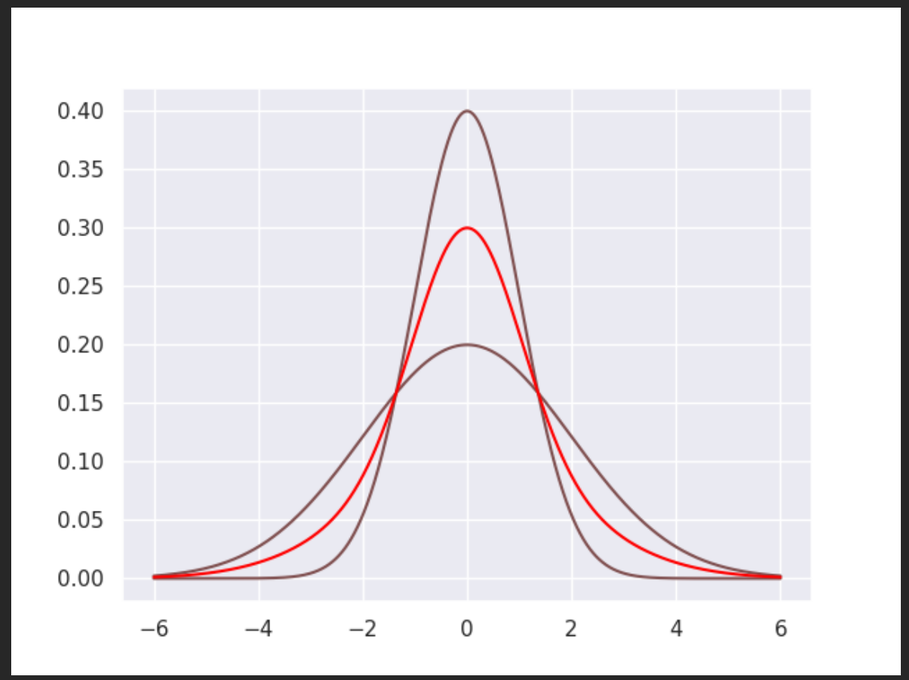
4. 乱数ライブラリー(C++11) (応用例)

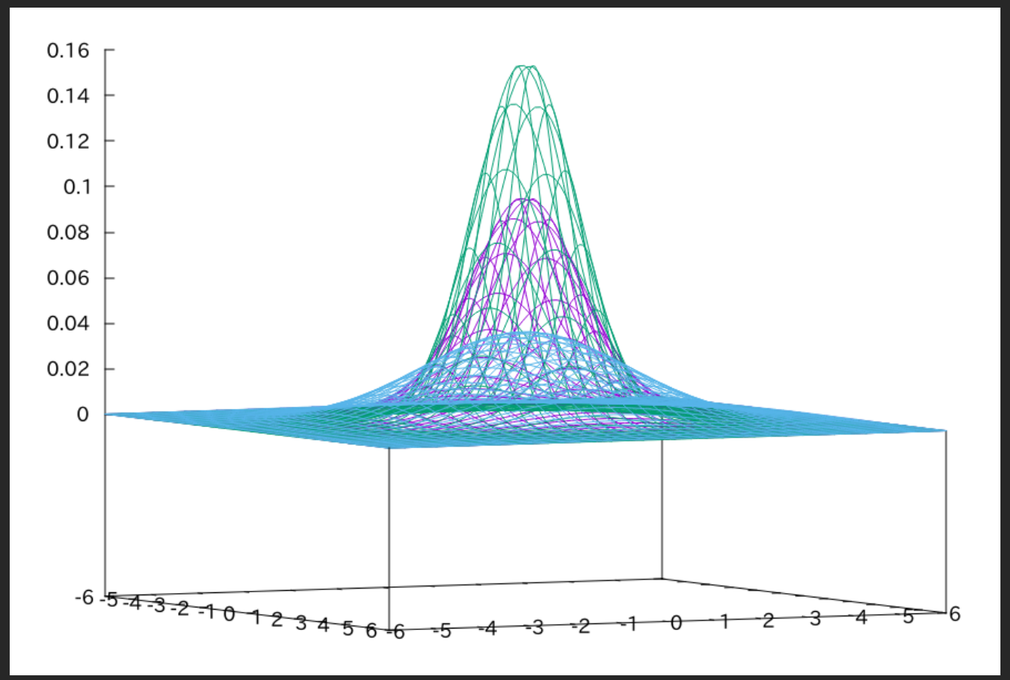
- 2つの 2次元正規分布で配置

今回はこうあって欲しい。中心にたくさんで、裾も広く。

4. 乱数ライブラリー(C++11) (応用例)

- 複数の正規分布の合成
  - 中心が鋭く、裾も広くなる。[Kawase 2004]
  - 先の例は、σの乱数、2σの乱数を半分ずつ生成。
    → 画像フィルター系の研究事例は参考になります。
    目的に合った分布を探しましょう!





σ=1とσ=2とそれらを足して2で割ったグラフ。1次元と2次元版。

※絵の出るスライドは以上です。

写真の草は、プログラムの参考にした小戸公園

5. ラムダ式のフル記述(C++17版)

- 先ほども出てきたラムダ式
  - 使わない機能は記述を省略できます。
  - 最小はキャプチャと本体。

```
// 最小ラムダ式
auto min_lambda = []                                    // キャプチャ
                  { printf("lambda min spec\n"); };     // 本体
min_lambda();

[実行結果]
lambda min spec
```

5. ラムダ式のフル記述(C++17版)

- 最大仕様のラムダ式(C++17時点)

auto ラムダ式 = [キャプチャ] (引数) [[属性]] mutable constexpr 例外仕様
-> 返値の型 { 本体 };

```
int    outside1 = 40;
double outside2 =  0.195;
auto full_lambda = [&outside1, outside2]          // キャプチャ
                   (int param1, double param2)    // 引数
                   [[nodiscard]]    // 属性       (nodiscard は C++17 から)
                   mutable          // mutable    (ラムダ式はデフォルトが const)
                   /*constexpr*/    // constexpr (明示せずとも可能な場合は constexpr になる)
                   noexcept(false)  // 例外仕様   (throw での記述は C++17 以降エラー)
                   -> double        // 返値の型
                   {                // 本体
                     outside1 += 1;
                     outside2 += 1.0;
                     if (outside1 == 42) throw outside1;
                     return (outside1 * param1) + (outside2 * param2);
                   };
```

5. ラムダ式のフル記述(C++17版)

- 最大仕様を記述をメモしておくと便利です。
- 先のラムダ式の呼び出しコードと結果。

```
printf("outside %d %f\n", outside1, outside2);
printf("lambda full spec %f\n", full_lambda(1, 1.0));
printf("outside %d %f\n", outside1, outside2);
try {
  [[maybe_unused]] double r = full_lambda(1, 1.0);
  // [[nodiscard]] 回避で返値をとるが、その unused 警告回避のための属性。
} catch(int e) {
  printf("exception %d\n", e);
}

[実行結果]
lambda min spec
outside 40 0.195000
lambda full spec 42.195000
outside 41 0.195000     <- outside1 は参照キャプチャなので、元の値も更新される
exception 42            <- outside1 == 42 で例外
```

6. 属性(C++17版)

- 関数や型への修飾、コンパイラへの指示

  各コンパイラが独自実装していましたが、C++11から言語標準で定義されるようになりました。

  - [[noreturn]] (C++11)

    関数が返らないことを示す。

    ※C++11 ではもう一つ [[carries_dependency]] が定義されましたが、大元の std::memory_order_consume が C++17 で一時的に非推奨となりましたので、一時的に気にしなくて構わないと思います。

  - [[deprecated]] (C++14)

    今後廃止予定であることを示す。

  - [[nodiscard]] (C++17)

    関数の返値を捨ててはいけないことを示す。

  - [[maybe_unused]] (C++17)

    オブジェクトが未使用で構わないことを示す。

6. 属性(C++17版)

- [[fallthrough]] (C++17)

  switch 内、case の処理が次の case に流れることを示す。

- [[noreturn]], [[deprecated]] の例

```
[[noreturn]] void noreturn_sample()
{
  printf("exit!\n");
  exit(0);
}

[[deprecated("もう使わないでね")]] void deprecated_sample()
{
  printf("obsolete func.\n");
}
```

6. 属性(C++17版)

- [[fallthrough]] +α の例(準備)

```
// サイコロ表示用
constexpr const char *dice_chara [6] = {
  u8"⚀", u8"⚁", u8"⚂", u8"⚃", u8"⚄", u8"⚅",// UTF-8 文字列定数
};

// 0 から 5 までを発生させるサイコロ
auto get_0origin_dice = [] {
  static std::default_random_engine gen(std::random_device{}());
  return std::uniform_int_distribution<>(0, 5)(gen);
};

printf("サイコロ: ");
for (int i=0; i<10; i++) {
  printf("%s ", dice_chara[get_0origin_dice()]);
}
printf( “\n” );
```

[実行結果]
サイコロ: ⚄ ⚃ ⚀ ⚃ ⚁ ⚀ ⚄ ⚀ ⚃ ⚀

6. 属性(C++17版)

- [[fallthrough]] +α の例

```
// 1 から 6 までを発生させる特殊なサイコロ
auto get_1origin_dice = [&] () {
  // if, switch の (初期化; 条件式) (C++17)
  switch (int d = get_0origin_dice(); (d += 1)) {
  case 1:
    printf("(%d->6)", d);  // 6 の裏は 1 ではなく 6
      [[fallthrough]];     // コンパイラーに処理が次の case に流れることを伝える
    case 6:
      return 6;

    case 2:
      printf("(%d->5)", d);
      [[fallthrough]];     // もし case が連続する場合は、無くても警告は出ない
    case 5:
      return 5;

    case 3:
      printf("(%d->4)", d);
      [[fallthrough]];
    case 4:
      return 4;
    }
    //printf("(%d)", d);   // スコープの外から d は見えない
};
```

6. 属性(C++17版)

- [[fallthrough]] +α の例(実行)

```
printf("特殊な賽: ");
for (int i=0; i<10; i++) {
  printf("%s ", dice_chara[get_1origin_dice() - 1]);
}
printf("\n");
```

[実行結果]

サイコロ: ⚃ ⚃ ⚀ ⚃ ⚁ ⚀ ⚅ ⚀ ⚃ ⚀

特殊な賽: (3->4)⚃ ⚃ (1->6)⚅ (2->5)⚄ (1->6)⚅ (3->4)⚃ ⚃ (1->6)⚅ (3->4)⚃ ⚃

7. 構造化束縛(C++17)

- structured binding declaration
- pair, tuple，配列、クラスを分解して、各要素に割り付ける宣言。
- こんな感じ。

```
auto [a, b, c] = get_tuple();
```

- 複数の値を関数から返す、多値返却に便利。

7. 構造化束縛(C++17)

- 多値返却の例(C++17)

```
#include <tuple>
// tupleに値をまとめて返す
std::tuple<int, double, std::vector<std::string>> tuple_return_sample_17()
{
  std::vector<std::string> strs;
  strs.push_back("A quick");
  strs.push_back("brown fox");
  strs.push_back(u8"いろはにほへと");
  return { 42, 42.195, strs };  // tupleのコンストラクタ改良
}

// i, d, strs 個別に型の推定が行われる
[[maybe_unused]] auto [i, d, strs] = tuple_return_sample_17();
(void)d; // 属性指定しても gcc7.3.0 で警告
printf("tuple get C++17 [%d %s]\n" , i, strs[2].c_str());

[実行結果]
tuple get C++17 [42 いろはにほへと]
```

7. 構造化束縛(C++17)

- 多値返却の例(C++14まで)

```
std::tuple<int, double, std::vector<std::string>> tuple_return_sample_14()
{
  std::vector<std::string> strs;
  strs.push_back("jumps over");
  strs.push_back("the lazy dog.");
  strs.push_back(u8"ちりぬるを");
  return std::make_tuple(42, 42.195, strs);
}

auto r = tuple_return_sample_14();
// 必要な要素を個々に get で取り出す
auto d = std::get<1>(r);
auto strs = std::get<2>(r);
printf("tuple get C++14 [%f %s]\n" , d, strs[1].c_str());

[実行結果]
tuple get C++14 [42.195000 the lazy dog.]
```

8. コピーの省略(C++11, C++17)

- copy elision, RVO(return value optimization), NRVO(Named -)
  - 関数から値を返す場合に、コンパイラはコピーを省略して良い(C++11)。
  - コピーのコストが重いオブジェクトを返す場合に、処理の軽減が見込める。

- コピーの省略の保証(C++17)
  - そもそもコピー、ムーブ不可のクラスは、値を返すことができなかった。
  - C++17で prvalue (右辺値で一時オブジェクト)に限り、コピーの省略を保証。
    → 返せるようになりました。

8. コピーの省略(C++11, C++17)

- コピーの省略の保証の例(C++17)

```
// コピー、ムーブ不可なクラス
struct NonCopyable {
  // 大きなメンバー
  std::vector<std::pair<double, double>> sin_tab;
  void print() {
    for (auto& [t, s] : this->sin_tab) {
      printf("%f %f\n", t, s);
    }
  }

  NonCopyable(int n) {
    for (int i=0; i<n; i++) {
      double t = i*2*M_PI/n;
      double s = sin(t);
      this->sin_tab.push_back({ t, s });
    }
  }
  NonCopyable(const NonCopyable &)  = delete;
  NonCopyable(const NonCopyable &&) = delete;
};
```

```
// オブジェクトを構築して返す関数
NonCopyable get_non_copyable(int n)
{
  //コピーの省略が保証され、返すことが出来る
  return NonCopyable(n);
}

auto a = get_non_copyable(8);
a.print();
//auto b = a;   // error:
```

```
[実行結果]
0.000000 0.000000
0.785398 0.707107
1.570796 1.000000
2.356194 0.707107
3.141593 0.000000
3.926991 -0.707107
4.712389 -1.000000
5.497787 -0.707107
```

8. コピーの省略(C++11, C++17)

- コピーの省略の許容の例(C++11)

```
// コピー可能なクラス
struct Copyable {
  // 大きなメンバー
(省略)
  // コンストラクター、デストラクター追跡用
  Copyable() {
    printf("%s: ctor.\n", __func__);
  }
  Copyable(const Copyable& a) {
    printf("%s: copy ctor.\n", __func__);
    this->sin_tab = a.sin_tab;
  }
  Copyable(const Copyable&& a) {
    printf("%s: move ctor.\n", __func__);
    this->sin_tab = std::move(a.sin_tab);
  }
  ~Copyable() {
    printf("%s: dtor.\n", __func__);
  }
};
```

```
// オブジェクトを構築して返す関数
Copyable get_copyable(int n)
{
  Copyable r;
  // 構築処理
  for (int i=0; i<n; i++) {
    double t = i*2*M_PI/n;
    double s = sin(t);
    r.sin_tab.push_back({ t, s });
  }
#if 1
  return r;  // 処理系でコピーを省略可能
  // あくまで可能なだけで、コピー、ムーブ
  // コンストラクターを削除してしまうと
  // コンパイルエラー。
#else
  return std::move(r);  // move が発生する
#endif
}
```

8. コピーの省略(C++11, C++17)

- コピーの省略の許容の例(C++11)
  - 不用意にmoveを書くより、コンパイラーに任せてコピーの省略を。

```
{
auto c = get_copyable(4);
c.print();
auto d = c; // コピーコンストラクト
d.print();
}
```

[実行結果、値で返す場合]

```
Copyable: ctor.
0.000000 0.000000
1.570796 1.000000
3.141593 0.000000
4.712389 -1.000000
Copyable: copy ctor.
0.000000 0.000000
1.570796 1.000000
3.141593 0.000000
4.712389 -1.000000
~Copyable: dtor.
~Copyable: dtor.
```

[実行結果、move で返す場合]
[-fno-elide-constructors で抑制した場合も同じ動作]

```
Copyable: ctor.
Copyable: move ctor.   <- NRVO の邪魔になる
~Copyable: dtor.       <- NRVO の邪魔になる
0.000000 0.000000
1.570796 1.000000
3.141593 0.000000
4.712389 -1.000000
Copyable: copy ctor.
0.000000 0.000000
1.570796 1.000000
3.141593 0.000000
4.712389 -1.000000
~Copyable: dtor.
~Copyable: dtor.
```

9. 非静的メンバー変数の初期化(C++11)

- クラスの定義時に、メンバー変数に初期化処理を書ける(C++11)
  - 簡単で便利です。
  - () での初期化処理は不可(関数宣言と判別がつかない)。

```
struct Vertex {
  // = か {} で初期化処理を記述
  float       x = 1, y, z = 11;
  uint32_t    color;
  std::string label { "ラベルなし" };

  // メンバー初期化子で初期化する場合、上の初期化は実行されない
  Vertex(std::string label) : label(label) {
    printf("%s %f %s\n", __func__, this->x, this->label.c_str());
  }
  // コンストラクター内で上書きは、上の初期化が先に実行される
  Vertex() {
    printf("%s %f %s\n", __func__, this->x, this->label.c_str());
    this->x = 3;
  };
};
```

10. 指示付き初期化子(C99, C++2a)

- Designated Initializer(C99)、C++ では C++2a で採用予定。

  最後に少しだけ未来の話をします。

- 構造体や配列を初期化リストで初期化する際に、構造体のメンバーや、配列のインデックスを指示して、初期化する機能。

- メンバーすべて、配列すべてを書かなくてすむ。

- こんな感じ。

  Vertex v = { .x =1，.z = 3, };

- 指示しなかった値は 0, +0.0, NULL

  指示なしで最初だけ書いた時と同じ。

  int a[100] = { 1 }; // 残り全部0

10. 指示付き初期化子(C99, C++2a)

- 指示付き初期化子の例(C99)

```
typedef struct {
  float    x, y, z;
  uint32_t color;
} Vertex;

void designated_initializer_sample(const Vertex *p0)
{
  Vertex p1 = {
    .color = 0xff008040,  // 初期化子の指示の順番は問わない
    .y = 42,
  };
  printf("%f %f %f %x\n", p0->x, p0->y, p0->z, p0->color);
  printf("%f %f %f %x\n", p1.x, p1.y, p1.z, p1.color);

  // 配列の一部の指示も可
  float mat[3][3] = { [0][0] = 1, [1][1] = 1, [2][2] = 1, };
  for (int i=0; i<3; i++) {
    for (int j=0; j<3; j++) {
      printf("%f ", mat[i][j]);
    }
    printf("\n");
  }
}
```

10. 指示付き初期化子(C99, C++2a)

- 指示付き初期化子の例(C99)

```
// 呼び出し
designated_initializer_sample(&(Vertex){ .x=1, .y=2, .z=3 });

[実行結果]
1.000000 2.000000 3.000000 0
0.000000 42.000000 0.000000 ff008040
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

10. 指示付き初期化子(C99, C++2a)

- C++2a では制約が多い。特に次の二つ。
  - 指示とメンバーの宣言の順序が一致していること
  - 配列には非対応

    ※また非静的メンバー変数の初期化が使われていると初期化リストが使えません。

- それでもとてもありがたい。

  「～と、冗長である指示付き初期化子(designated initializer)以外のほとんどの新機能を取り込んだ」[プログラミング言語C++第４版, 1.2.3 C言語との互換性] と C++11 の時に書かれましたが、C++ の C との互換性がまた一歩進みました。

10. 指示付き初期化子(C99, C++2a)

- 指示付き初期化子の例(C++2a)

```
struct ForTheFuture {
  float    x, y, z;
  std::string label;
};
void designated_initializer_sample(const ForTheFuture &f1)
{
  ForTheFuture f0 = {
    .x = 89,  // メンバー宣言順と合わせる必要がある
    .y = 99,  // 順があっていれば途中は省略可能の予定
    .z = 11,  //  (gcc7.3.0ではコンパイルエラー)
    .label="明るい"
  };
  printf("%f %f %f %s\n", f0.x, f0.y, f0.z, f0.label.c_str());
  printf("%f %f %f %s\n", f1.x, f1.y, f1.z, f1.label.c_str());
}
designated_initializer_sample({.x=14, .y=17, .z=20, .label= “未来"});

[実行結果]
89.000000 99.000000 11.000000 明るい
14.000000 17.000000 20.000000 未来
```

まとめ

- 1. 数値定数(C++14, C99/C++17)
- 2. hypot(C99/C++17)
- 3. aligned_alloc(C11/C++17)
- 4. 乱数ライブラリ(C++11)
- 5. ラムダ式のフル記述(C++17版)
- 6. [[属性]] (C++17版)
- 7. 構造化束縛(C++17)
- 8. コピーの省略(C++11, C++17)
- 9. 非静的メンバー変数の初期化(C++11)
- 10. 指示付き初期化子(C99/C++2a)